Abstract — The importance of Aspect-Oriented Programming Systems testing techniques grows with the incremental acceptance of this paradigm by software engineers and designers. Despite introducing new advantages in software development, as programs become more cohesive and modular, AOP systems are quite hard to test because of the characteristics of the paradigm. In this paper we introduce those characteristics that make AOP systems hard to test and present some testing techniques.

Index Terms — Aspect Oriented Programming, Aspectual State Models, Fault-Based, Model-Based, Software Testing, Software Verification and Validation.

I. INTRODUCTION

Software Verification and Validation plays a significant role in the development of the final executable software, at the same time society is increasingly dependent on systems witch software its critical. Health care, financial, transportation, and military systems all rely on software to perform or support their functions and programs whose malfunction would have severe consequences justify greater effort in their validation. The increasing trend to use computers to solve complex problems has led to the construction of extremely complex software systems. In order to be maintained and also for reusability sake, those systems must be modularized. Programming paradigms such as procedural and object-oriented ones include modularization constructs (e.g. procedures, functions, classes, packages, etc). However, once a concern has been set, the resulting system will be tied to that concern.

OOP(Object-Oriented Programming) is not the best paradigm to describe crosscutting concerns over objects. Adding crosscutting concerns such as optimizations tends to cause tangling over a program, causing the program to become difficult to maintain and understand. AOP is proposed in order to overcome this problem.

In AOP, crosscutting concerns are factored into distinct abstractions called aspects, which are solely responsible for a particular crosscutting concern. This raises the system's modularity and increases cohesion of abstractions that are primary concerns.

In AOP systems, the processes of validation testing are no different that for any other system. The final executable program is treated as a black-box and tests are devised to show whether or not the system meets its requirements. However there are real problems with program inspections when AOP is used and when white-box testing is used in the source code to identify potential defect [1]:

- Aspects specification so that tests for these aspects may be derived.
- Aspects testing independence of the base system with which they should be woven.
- Testing of aspect interference (aspect interference occurs when two or more aspects use the same pointcut specification).
- Design of tests in such a way that all program join points are executed and appropriate aspects applied.

Regardless of the programming technology used, a primary goal is to produce high quality software. As a consequence, AOP systems must also be tested.

In order to answer to all these quality related questions, we should consider the following issues [3]:

1. Aspects do not have independent identify or existence: they depend upon the context of some other class for their identify and execution context.
2. Aspect implementations can be tightly coupled to their woven context: aspects depend on the internal representation and implementation of classes into which they are woven. Changes to these classes will likely propagate to the aspects.
3. Control and data dependencies are not readily apparent from the source code of aspects or classes: due to the nature of the weaving process, the developer of the classes or aspects knows neither the resulting control flow nor the data flow structure of the resulting woven artefact. Thus, relating failures to the corresponding faults may be difficult.
4. Emergent behaviour: the root cause of a fault may lie in the implementation of a class or an aspect, or it may be a side effect of a particular weave order of multiple aspects.
The remainder of this paper is organized as follows: in Section II we present the methodology used in our survey, that is, how we made our research, what motivated us and what was used as our literary support; in Section IV we discuss four AOP systems testing techniques, their differences, advantages and disadvantages.

II. UNIT TESTING

Unit (module or component) level testing focuses on the early examination of sub-program functionality and ensures that functionality not visible at the system level is examined by testing. [14] Unit testing ensures that quality software units are furnished for integration into the finished software product. [14]

Unit testing is a fundamental practice in most software development methodologies, including Extreme Programming. Unit testing is usually performed before integration testing, to detect faults that pertain to the units, independent of the rest of the system.

While aspect-oriented programming improves modularity by allowing the encapsulation of crosscutting concerns into units, testing such units poses significant challenges. [15]

Many techniques for aspect-oriented unit testing based on dynamic behaviour were propose. Zhao [16] has proposed a data flow based approach to unit testing of aspect-oriented programs. For each aspect or class, the approach performs three levels of testing, i.e., intra-module, inter-module, and intra-aspect/intra-class testing.

Lemos et al. [17] proposed a control and data flow model for AO programs based on AspectJ, and defined a family of testing criteria and tool to unit test these programs [15].

Many frameworks were presented to solve problems in this area. JUnit and mock object [18] are used in different tools and presented proposes. JUnit is a widely used framework for unit-testing Java programs. Cristina Videira Lopes and Trung Chi Ngo [15] approach is based on JAML, an extensible language framework that they have developed. In [15] they briefly describe JAML, and they introduce JamlUnit, the proposed approach towards unit-testing aspects written in JAML.

JamlUnit consists of a set Java helper classes that enable programmers to write regular JUnit test cases for testing aspect code as independent units.

An unit testing framework for aspects without weaving [19] is present to resolve problems previous presented in the unit testing research area and the combination with other testing frameworks is considered, which reveal the numerous hypotheses and open issues in this area.

III. MODEL-BASED TESTING

According to [2], Model-Checking is a formal method that checks whether a structure (system) satisfies a property or not. Model checking has been used in order to verify the correctness of concurrent finite automata including LSI design and communication protocols. Model checking will be useful for AOP validation because it is used to check global properties across multiple processes. The paper suggests the application of model checking to verify automatically whether a program composed of aspects and objects satisfies properties specified as temporal logic formula that describes requirements specifications such as error logging, safety, liveness, fairness and so on.

Some model checkers are presented in the paper, such as SPIN [11] and SMV [12] Java PathFinder [9], Bandera [10] and ESC/Java [13] are specialized tools for checking properties of Java programs.

An AOP-based model checking framework is proposed in order to use model checking tools efficiently, which can be used not only with model checking but also with runtime testing. The approach proposed has a few advantages:
- Readability is improved because checking properties can be separated from original functions.
- Checking properties that crosscut over objects can be encapsulated into an aspect.
- It is not necessary to modify program code whenever new checking features are added.
- It is only necessary to weave aspects that describe the checking features.
- Aspects are defined corresponding to checking viewpoints.

Both a unit testing that weaves only one aspect and an integrated testing that weaves multiple aspects are possible. Although, problems such as performance and memory efficiency are unavoidable to model checking. Because model checkers such as Jpf [9] and Bandera [10] adopt source code as a model, the space explosion problem cannot be ignored.

IV. FAULT-BASED TESTING

Roger, James and Anneliese [3] introduce a candidate fault model along with associated testing criteria for AOPs based on interactions that are unique to AOP. They also identify key issues relevant to the systematic testing of AOPs.

According to [3], a fault model for AOP should reflect those characteristics that are distinct from object-oriented and procedure-oriented programming. A key issue is to identify the fault types that are unique to AOP systems. Because AOP is a new technology, new fault types have not been identified from practical experience. The paper suggests that there are four potential sources of faults in a program with woven aspects:

1. The fault resides in a portion of the core concern that is not affected by a woven aspect.
2. The fault resides in code that is specific to the aspect, isolated from the woven context.
3. The fault is an emergent property created by interactions between one aspect and the primary abstraction.
4. The fault is an emergent property created when more than one aspect is woven into the primary abstraction.

The candidate fault model is based on the peculiarities of AOP systems and a testing criteria is derived from the candidate fault model. In [4], Mariano, Paolo and Filippo briefly summarize and slightly extend the fault model by Alexander et al. [3] with a couple of additional fault types, which will constitute the basis for the further discussion presented in their paper.
V. STATE-BASED TESTING

Traditionally one commonly used testing technique is to analyze the different abstract states that a class can take. The state of an object is generally defined as a constraint on the values of its attributes. According to the state of the object, calls to certain methods may or may not be valid, or the method’s behaviour may change.

In Aspect-Oriented Software Development (AOSD), validating whether or not the aspects of crosscutting concerns are implemented correctly is a major issue. It requires adequate exercise of classes as well as aspects [20]. Xu et al. [20] describe a state-based testing approach to test AOP programs. They present a model call Aspectual State Model (ASM), which is based on the know FREE (Flattened Regular Expression) state model. Their ASM can derive test cases for the scenario of a class weaved by multiple aspects, this technique transforms an aspectual state model to a transition tree and generates tests based on the tree.

The purpose of the method is to validate dynamic behaviour of objects, including interactions, dependencies, and constraints on the message sequences in terms of the state model [20].

In the approach proposed by Xu et al. [20], the set (class and aspect) is treated as a unique block. It is therefore difficult to localize an error when it occurs. By integrating aspects incrementally, the complexity of the test is reduced. In case of failure is possible target the origin of the error and another approach for generating unit test sequences for aspect-oriented programs is proposed by Bradi et. al [21].

However the problem of integration aspects is not that simple, Xu et al [22] presented another effort to resolve problems and a state-base incremental testing of AOP process is presented. The incremental testing approach is similar to traditional regression testing. The essential difference is that, aspects as a structured way to specify modifications make it feasible to investigate systematic reuse and modification of the existing tests [22].

VI. CRITERIA FOR COMPARING TESTING TECHNIQUES

In this section we will present a comparison table of all the techniques described so far, using the following criteria: automation (whether there is a tool for automate the process or not), target phase (in which phase(s) should the technique be applied: design, programming, execution), maturity (proposa, experimental or adoption), AOP specific, effectiveness and efficiency.

<table>
<thead>
<tr>
<th></th>
<th>Unit Testing</th>
<th>Model-Based Testing</th>
<th>Fault-Based Testing</th>
<th>State-Based Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Automation</td>
<td>Research Tool</td>
<td>Research Tool</td>
<td>Research Tool</td>
<td>Research Tool</td>
</tr>
<tr>
<td>Target Phase</td>
<td>Implementation</td>
<td>Design, Programming</td>
<td>Execution</td>
<td>Execution</td>
</tr>
<tr>
<td>Maturity</td>
<td>Proposal</td>
<td>Experimental</td>
<td>Proposal</td>
<td>Proposal</td>
</tr>
<tr>
<td>AOP Specific</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

VII. CONCLUSIONS

The model-checking approach proposed by Naoyasu and Tetsuo [2] shows effectiveness, despite the existence of problems that should be resolved. According to Naoyasu and Tetsuo, it is important to check individual aspects and objects when programs woven from the aspects and objects are large. In order to solve this problem, they suggest an approach such that model checking is applied to a result of weaving an aspect and stub objects generated from the aspect.

The fault-model proposed by Roger, James and Anneliese in [3] is a first important step in developing an effective approach to the systematic testing of AOPs, although it still needs to be empirically validated and possibly extended and refined. Roger, James and Anneliese suggest that there are still open questions that need to be answered:

1. Are there ways to test aspects on their own?
2. Can we reverse engineer the weave process?
3. Can we measure test coverage after weaving?
4. How do we test aspects that interact with a core concern?
5. How do we test aspects that mutually interfere?
6. How do we test aspects whose effects must span more than one concern?

Many researchers are investigating techniques for reasoning about aspect-oriented systems, with the goal of understanding and verifying the effects of aspects on the underlying primary concerns as well as aspect interactions [23]. Although many types of errors can be detected, in general we cannot prove an aspect-oriented program is correct, and so testing complements formal and informal approaches to verification [23]. There are still open questions related to testing AOP Systems that have not been answered yet. However aspect-oriented software development has gained popularity with the adoption of languages such as AspectJ and at the same time there is significant effort to development testing frameworks for AOP.

VIII. REFERENCES

[14] General Principles of Software Validation; Final Guidance for Industry and FDA Staff